# ELFIO

## Tutorial

**Allan Finch**

**Serge Lamikhov-Center**

**ELFIO: Tutorial**

by Allan Finch  and Serge Lamikhov-Center

# Table of Contents

# Introduction

ELFIO is a C++ library for reading and generating files in the ELF binary format. This library is unique and not based on any other product. It is also platform independent. The library uses standard ANSI C++ constructions and runs on a wide variety of architectures.

While the library's implementation does make your work easier: a basic knowledge of the ELF binary format is required. Information about ELF is included in the TIS (Tool Interface Standards) documentation you received with the library's source code.

# Chapter 1. Getting Started With ELFIO

## Initialization

The ELFIO library consists of two independent parts: ELF File Reader (IELFI) and ELF Producer (IELFO). Each is represented by its own set of interfaces. The library does not contain any classes that need to be explicitly instantiated. ELFIO itself provides the interfaces that are used to access the library's functionality.

To make the program recognize all ELFIO interface classes, the ELFIO.h header file is needed. This header file defines all standard definitions from the TIS documentation.

```
#include <ELFIO.h>
```

This chapter will explain how to work with the reader component of the ELFIO library. The first step is to get a pointer onto the ELF File Reader:

```
IELFI* pReader;
ELFIO::GetInstance()->CreateELFI( &pReader );
```

Now, that there is a pointer on the IELFI interface: initialize the object by loading the ELF file:

```
char* filename = "file.o";
pReader->Load( filename );
```

From here, there is access to the ELF header. This makes it possible to request file parameters such as encoding, machine type, entry point, etc. To get the encoding of the file use:

```
unsigned char encoding = pReader->GetEncoding();
```

Please note: standard types and constants from the TIS document are defined in the ELFTypes.h header file. This file is included automatically into the project. For example: ELFDATA2LSB and ELFDATA2MSB constants define a value for little and big endian encoding.

## ELF File Sections

ELF binary files consist of several sections. Each section has it's own responsibility: some contain executable code; others describe program dependencies; others symbol tables and so on. See the TIS documentation for a full description of each section.

To see how many sections the ELF file contains, including their names and sizes, is demonstated in the following code:

```
int nSecNo = pReader->GetSectionsNum();
for ( int i = 0; i < nSecNo; ++i ) {    // For all sections
    const IELFISection* pSec = pReader->GetSection( i );
    std::cout << pSec->GetName() << " "
              << pSec->GetSize() << std::endl;
    pSec->Release();
}
```

First, the number of sections are received; next, a pointer on the IELFISection interface. Using this interface, access is gained to the different section attributes: size, type, flags and address. To get a buffer that contains the section's bytes use the `GetData()` member function of this interface. See the IELFISection declaration for a full description of the IELFISection interface.

# Section Readers

After the section data is received through the `GetData()` function call, the data can be manipulated. There are special sections that provide information in predefined forms. The ELFIO library processes these sections. The library provides a set of section readers that understand these predefined formats and how to process their data. The ELFIO.h header file currently defines the types of readers as:

```
enum ReaderType {
    ELFI_STRING,     // Strings reader
    ELFI_SYMBOL,     // Symbol table reader
    ELFI_RELOCATION, // Relocation table reader
    ELFI_NOTE,       // Notes reader
    ELFI_DYNAMIC,    // Dynamic section reader
    ELFI_HASH        // Hash
};
```

How to use the symbol table reader will be demonstated in the following example:

First, get the symbol section:

```
const IELFISection* pSec = pReader->GetSection( ".symtab" );
```

Second, create a symbol section reader:

```
IELFISymbolTable* pSymTbl = 0;
pReader->CreateSectionReader( IELFI::ELFI_SYMBOL,
                              pSec,
                              (void**)&pSymTbl );
```

And finally, use the section reader to process all entries (print operations are omitted):

```
std::string   name;
```

```
Elf32_Addr    value;
Elf32_Word    size;
unsigned char bind;
unsigned char type;
Elf32_Half    section;
int nSymNo = pSymTbl->GetSymbolNum();
if ( 0 < nSymNo ) {
    for ( int i = 0; i < nSymNo; ++i ) {
        pSymTbl->GetSymbol( i, name, value, size,
                            bind, type, section );
    }
}
pSymTbl->Release();
pSec->Release();
```

# Finalization

All interfaces from the ELFIO library should be freed after use. Each interface has a `Release()` function. It is not enough to only free the high level interface because one of the sections or readers will still be held and its resources will not be cleared.

The interfaces are freed immediately after their use, in this example we will free only the pReader object:

```
pReader->Release();
```

# Chapter 2. ELFDump Utility

The source code for the ELF Dumping Utility can be found in the "Examples" directory; included there are more examples on how to use different ELFIO reader interfaces.

# Chapter 3. IELFO - ELF File Producer Interface

The ELFIO library can help you build a very short ELF executable file. This chapter shows how to build an executable file that will run on x86 Linux machines and print "Hello World!" on your console.

Just as with the reader, the first step is to get a pointer onto the ELF File Writer (Producer):

```
IELFO* pELFO;
ELFIO::GetInstance()->CreateELFO( &pELFO );
```

Before continuing, the library must be informed about the main attributes of the executable file to be built. To do this, declare that the executable ELF file will run on a 32 bit x86 machine; has little endian encoding and uses the current version of the ELF file format:

```
// You can't proceed without this function call!
pELFO->SetAttr( ELFCLASS32, ELFDATA2LSB, EV_CURRENT,
                ET_EXEC, EM_386, EV_CURRENT, 0 );
```

Some sections of an ELF executable file should reside in the program segments. To create this loadable segment call the `AddSegment()` function.

```
// Create a loadable segment
IELFOSegment* pSegment = pELFO->AddSegment( PT_LOAD,
                                            0x08040000,
                                            0x08040000,
                                            PF_X | PF_R,
                                            0x1000 );
```

The following segment serves as a placeholder for our code section. To create this code section call the AddSection() function:

```
// Create code section
IELFOSection* pTextSec = pELFO->AddSection( ".text",
                                SHT_PROGBITS,
                                SHF_ALLOC | SHF_EXECINSTR,
                                0,
                                0x10,
                                0 );
```

Then, add the executable code for the section:

```
char text[] =
{ '\xB8', '\x04', '\x00', '\x00', '\x00',   // mov eax, 4
  '\xBB', '\x01', '\x00', '\x00', '\x00',   // mov ebx, 1
  '\xB9', '\xFD', '\x00', '\x04', '\x08',   // mov ecx, msg
  '\xBA', '\x0E', '\x00', '\x00', '\x00',   // mov edx, 14
  '\xCD', '\x80',                           // int 0x80
```

```
    '\xB8', '\x01', '\x00', '\x00', '\x00',   // mov eax, 1
    '\xCD', '\x80',                           // int 0x80
    '\x48', '\x65', '\x6C', '\x6C', '\x6F',   // db 'Hello'
    '\x2C', '\x20', '\x57', '\x6F', '\x72',   // db ', Wor'
    '\x6C', '\x64', '\x21', '\x0A'            // db 'ld!', 10
};
pTextSec->SetData( text, sizeof( text ) );
```

Next, this code section is put into the loadable segment:

```
// Add code section into program segment
pSegment->AddSection( pTextSec );
pTextSec->Release();
pSegment->Release();
```

Finally, define the start address of the program and create the result file:

```
// Set program entry point
pELFO->SetEntry( 0x08040000 );
// Create ELF file
pELFO->Save( "test.elf" );
pELFO->Release();
```

Please note: Call the `Release()` functions for each interface you have used. This will free all resources the ELFIO library has created.

Now compile the program and run it. The result is a new ELF file called "test.elf". The size of this working executable file is only 267 bytes! Run it on your Linux machine with the following commands:

```
[Writer]$ ./Writer
[Writer]$ chmod +x test.elf
[Writer]$ ./test.elf
Hello, World!
```

The full text for this program can be found in the "Writer" directory. Also, in the "Examples" directory, two other programs "WriteObj" and "WriteObj2" demonstrate the creation of ELF object files.